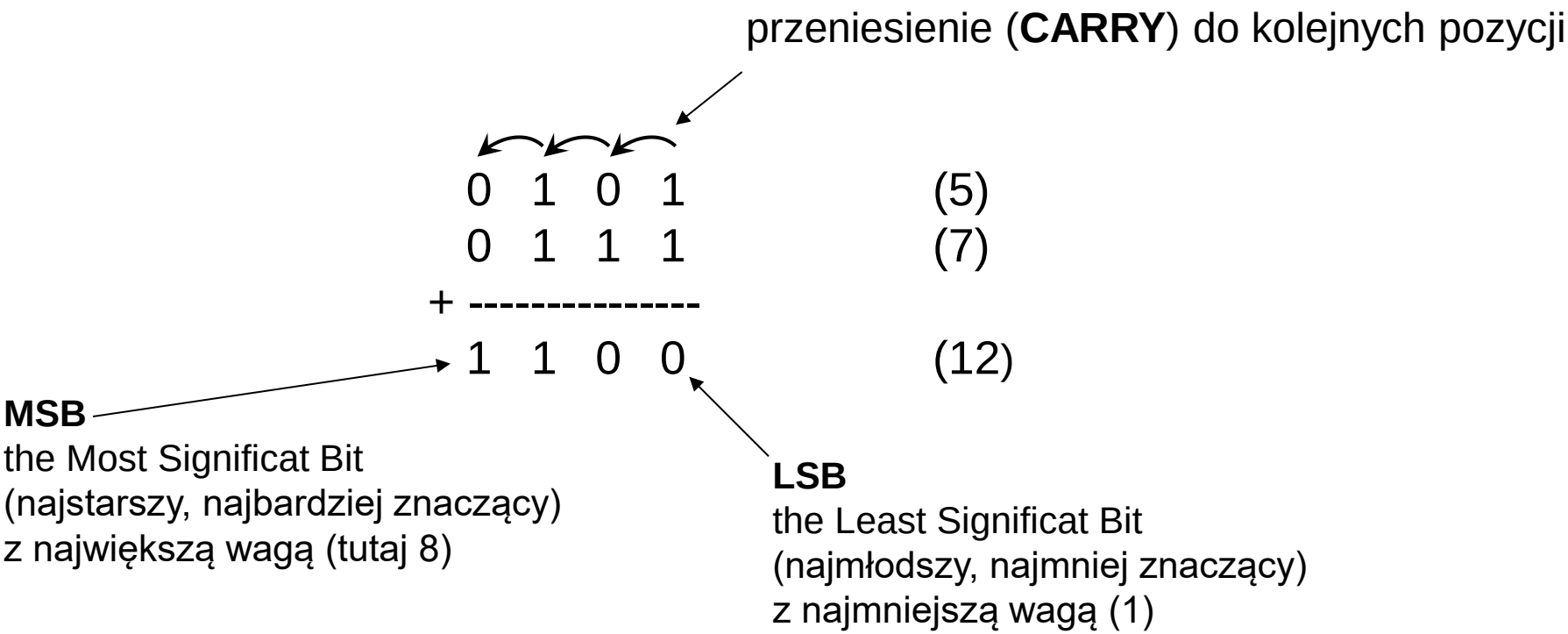
Arytmetyka komputerowa

- dodawanie,
- odejmowanie / kod U2 (uzupełnień do dwóch),
- przesunięcia bitowe i arytmetyczne,
- rozszerzenie długości słowa,
- operacje na bitach testowanie/ustawianie/kasowanie/zmiana,
- mnożenie,
- dzielenie.

Dodawanie

w systemie binarnym jest trywialne…



- przykład – półbajty / nibbles (4 bity, bez znaku: zakres od 0 do 15)
  niemniej zasady przeprowadzania operacji są takie same dla dłuższych słów

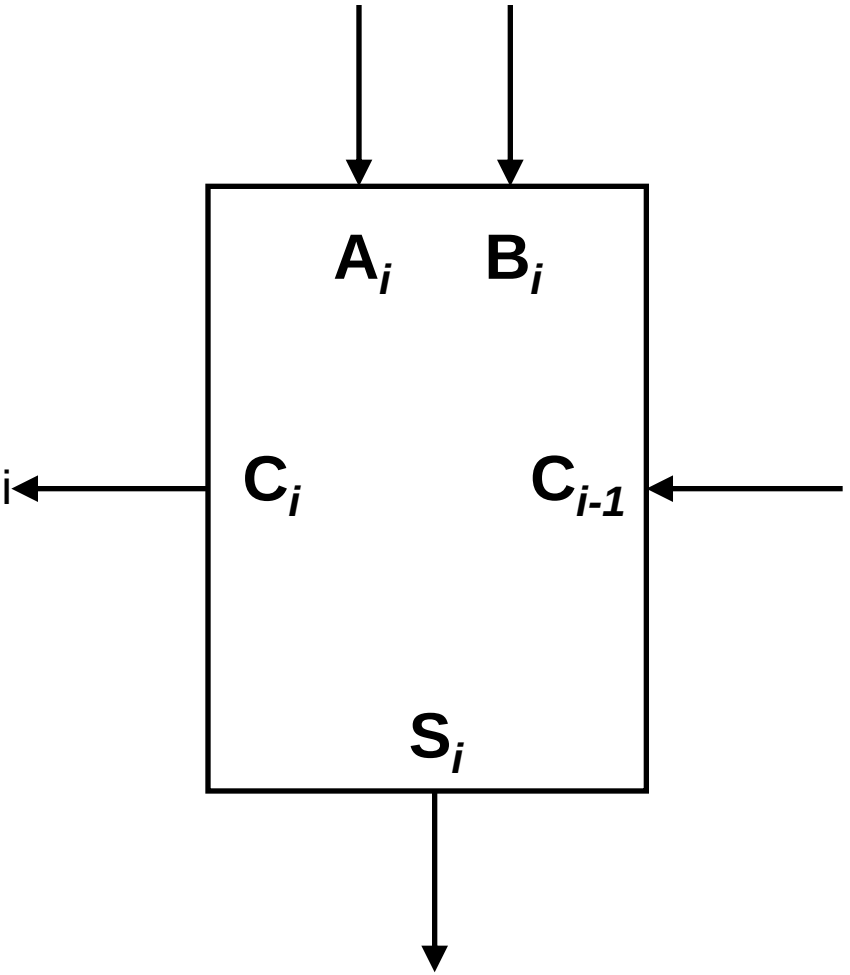
Sumator pełny (Full Adder - FA)
(jednobitowy)

A_i, B_i – wejścia *i*-tch bitów dodawanych argumentów

C_{i-1} – wej. przeniesienia z poprzedniej pozycji

C_i - wyj. przeniesienia do następnej pozycji

S_i – *i*-ty bit wyniku (sumy)



• tzw. **półsumator** (Half Adder - HA) nie posiada wejścia C_{i-1}

Sumator pełny

tablica prawdy

wszystkie kombinacje zmiennych wejściowych i odpowiadające im stany wyjść:

C_{i-1}	B_i	A_i	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- stan każdego wyjścia określa funkcja logiczna* :

$$C_i = A_i B_i + C_{i-1} B_i + C_{i-1} A_i$$

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

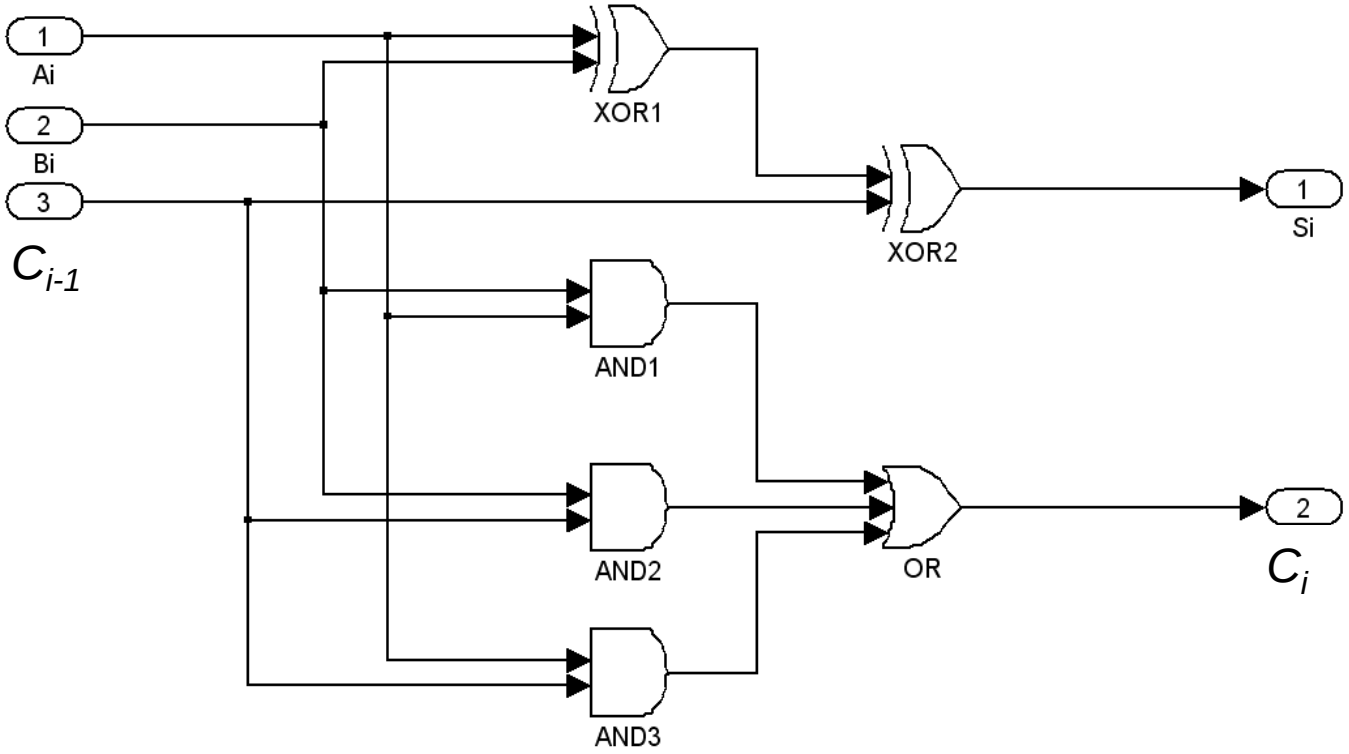
- funkcje te można zapisać bezpośrednio na podstawie tablicy prawdy, a następnie minimalizować na drodze przekształceń algebraicznych, bądź używając metod np. graficznych (tablice Karnaugha).

Sumator pełny (1-bit)

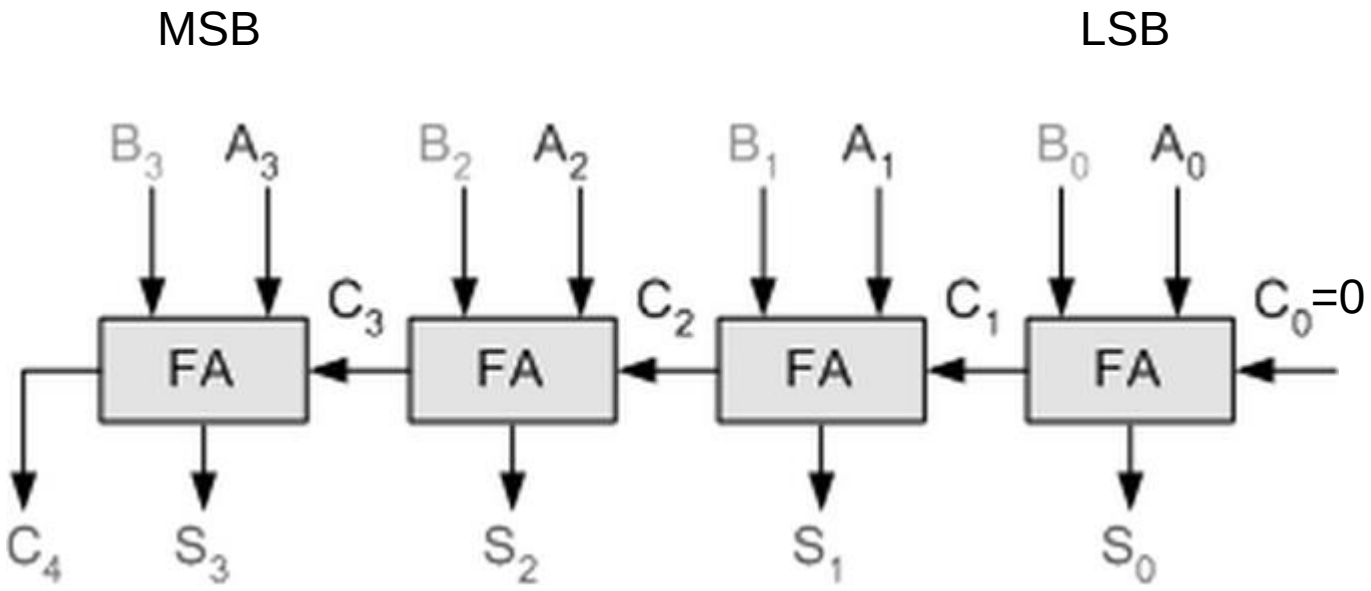
przykładowe (jedno z możliwych) rozwiązanie układowe:

$$C_i = A_i B_i + C_{i-1} B_i + C_{i-1} A_i$$

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$



Prosty sumator wielobitowy



czas propagacji sygnału przeniesienia: *2nt*

(dla układu jednobitowego z poprzedniego schematu)

n – liczba bitów w słowie, *t* - czas propagacji jednej bramki logicznej

Kod uzupełnień do 2 - U2 (two's complement)

kodowanie liczb ze znakiem

• różnica między naturalnym kodem binarnym (NB):

waga najstarszej pozycji ze znakiem minus

$$N_{U2} = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad a_i \in \{0,1\}$$

np.
półbajt (4 bity) wagi **8** 4 2 1 (NB) zakres: 0,…,15
-8 4 2 1 (U2) zakres: -8,…,7

bajt bez znaku: wagi **128**, 64,…, 1 zakres: 0 ,…, 255
bajt ze znakiem: **-128**, 64,…, 1 zakres: -128,…,127

• nie ma „strat": podwójnego zera (-0, +0), wykorzystane wszystkie kombinacje „0" i „1",

• MSB oprócz wartości określa również znak liczby,

• ten sam ciąg „zer i jedynek" można interpretować jako różne liczby, np.:
„1001" jako 9 (NB) lub -7 w U2…

Odejmowanie, czyli…

$$A - B = A + (-B)$$

Jak obliczyć „-*B*” ?

$$B + (-B) = 0 \qquad \text{(x)}$$

z własności algebry boolowskiej:

$$B + \overline{B} = 1$$

analogicznie, dla słów wielobitowych:

$$B + \overline{B} = 1 \ldots 1$$

+1 do obu stron:

$$B + \overline{B} + 1 = 0 \qquad \text{(y), przeniesienie z MSB zaniedbać…}$$

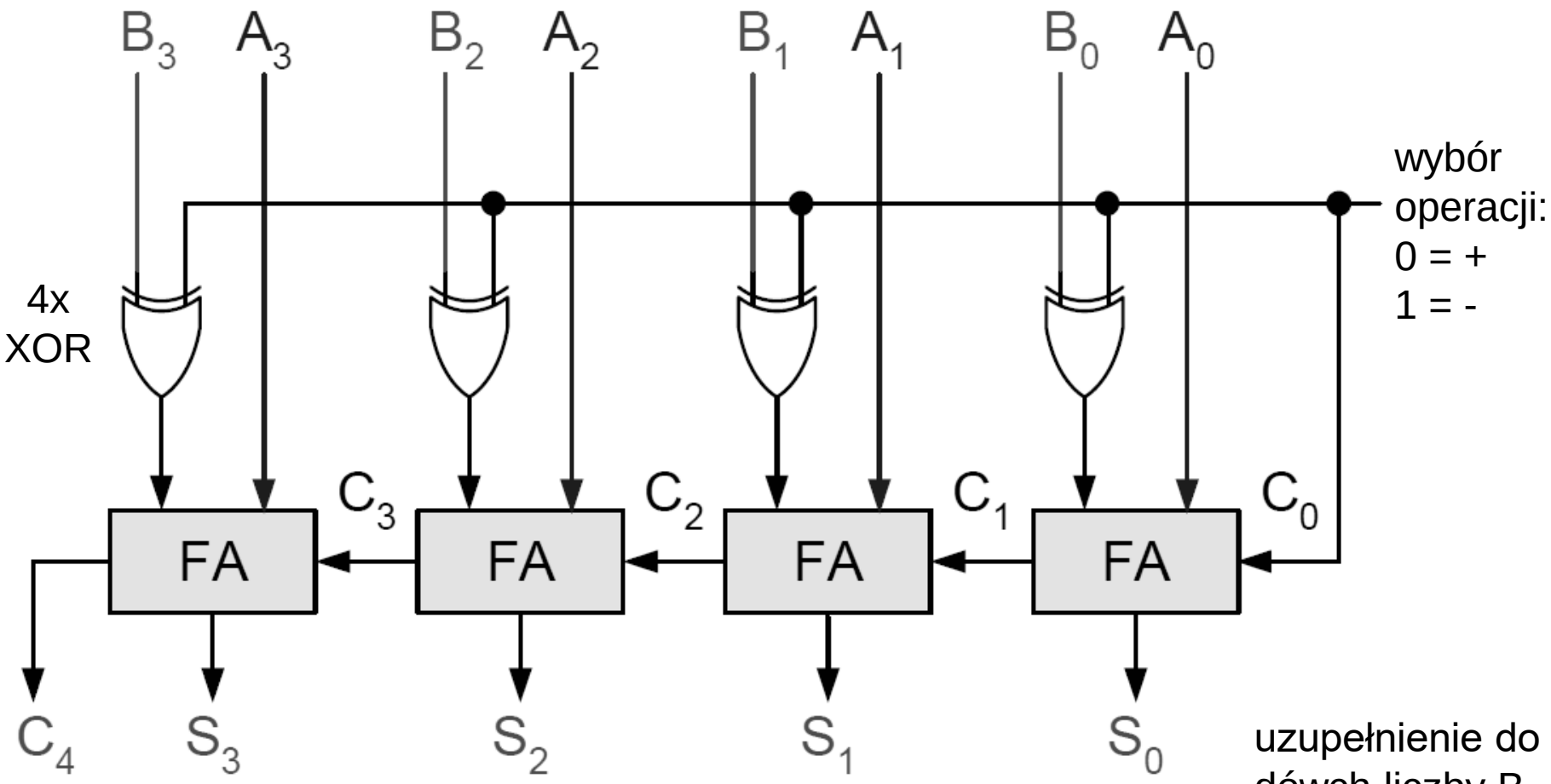
porównując (x) z (y) otrzymujemy **uzupełnienie do dwóch liczby *B*** (two's complement):

$$-B = \overline{B} + 1$$

- **w arytmetyce komputerowej odejmowanie *A - B* przeprowadza się poprzez dodanie uzupełnienia do dwóch liczby *B:***

$$A - B = A + \overline{B} + 1$$

Sumator bardzo łatwo rozbudować o możliwość wykonywania drugiej operacji - odejmowania…



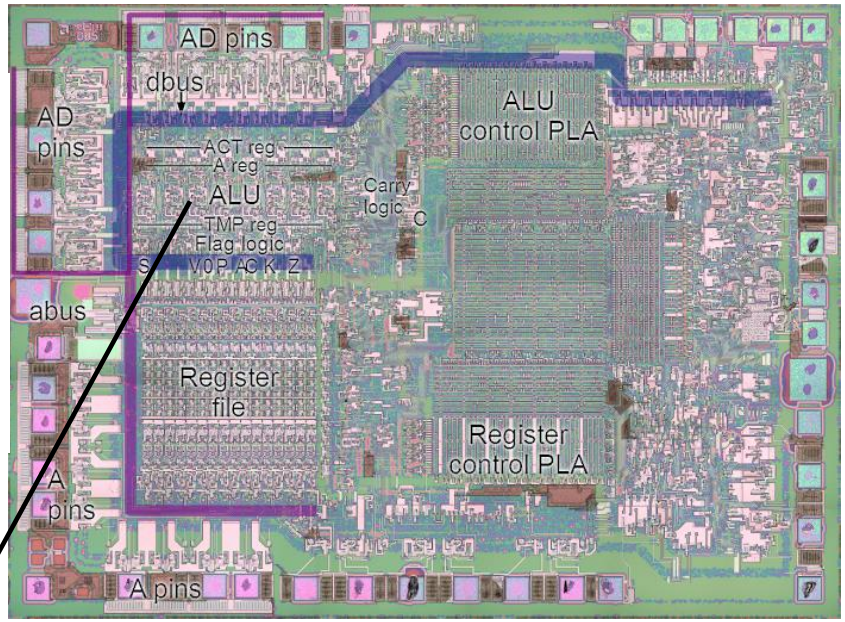
$$A - B = A + \overline{B} + 1$$

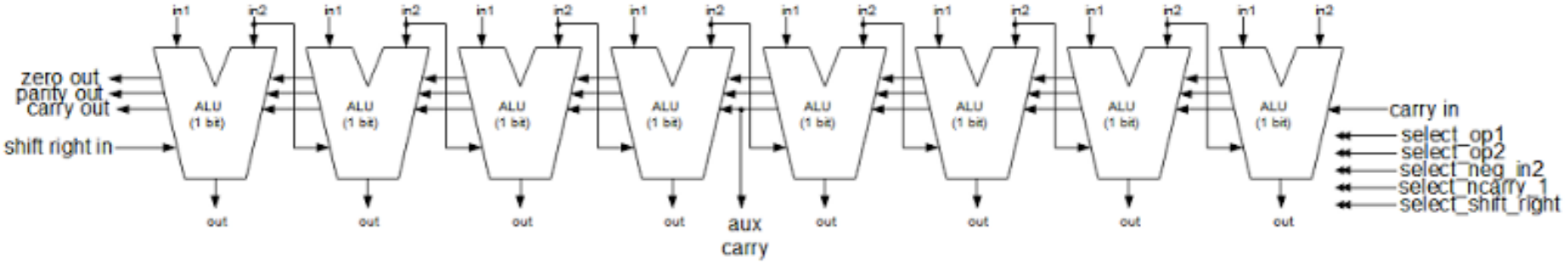
$$-B = \overline{B} + 1$$

Jednostka arytmetyczno logiczna (ALU) 8-bitowego procesora Intel 8085

www.righto.com/2013/07/reverse-engineering-8085s-alu-and-its.html

Operation	*select_neg_in2*	*select_op1*	*select_op2*	*select_shift_right*	*select_ncarry_1*	Carry in/out
or	0	0	0	0	1	1
add	0	1	0	0	0	/carry
xor	0	1	0	0	1	1
and	0	1	1	0	1	0
shift right	0	0	1	1	1	0
complement	1	0	0	0	1	1
subtract	1	1	0	0	0	borrow





Działania na liczbach bez znaku

- **poprawność obliczeń**
- **przekroczenie zakresu**

przykład: dodawanie półbajtów (zakres: 0…15)

C_{MSB}=0 ← 0110 (6)
0111 (7)
\+ -------
1101 (13) OK

C_{MSB}=1 ← 0111 (7)
1101 (13)
\+ -------
0100 (4) ?? (tylko 4 bity)

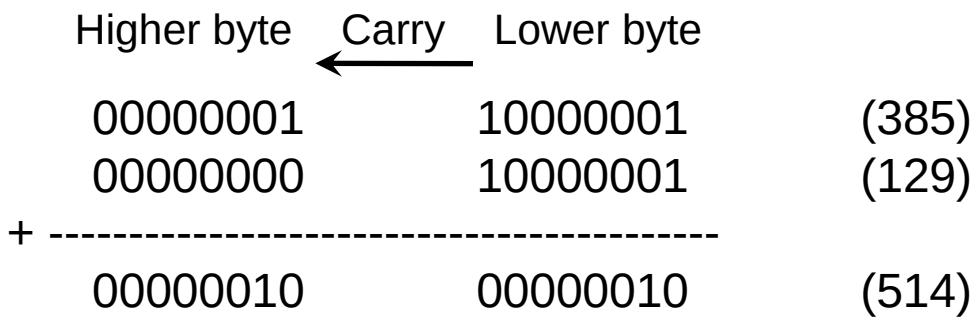
10100 (16+4) OK (potrzeba 5 bitów)

Interpretacja przeniesienia (CARRY) z najstarszej pozycji (MSB) podczas dodawania liczb bez znaku jest prosta i jednoznaczna:

- wystąpienie tego przeniesienia oznacza przekroczenie zakresu,

- sam wynik jest błędny, ale sygnał przeniesienia można wykorzystać do obliczeń na dłuższych typach danych.

Dodawanie „dłuższych” typów danych:



np. procesor 32-bitowy (80386):

1. liczba 64-bitowa w parze %ebx : %eax (Higher : Lower)
2. liczba 64-bitowa w parze %edx : %ecx (Higher : Lower)

```
ADD     %ecx , %eax
ADC     %edx , %ebx
```

rozkaz ADC = ADd with Carry – dodaj z uwzględnieniem przeniesienia z poprzedniej operacji.

64-bitowy wynik znajduje w parze rejestrów %ebx : %eax (Higher : Lower).

- operacja musi być przeprowadzona etapami (na fragmentach danych odpowiadających szerokości ALU/rejestrów),
- wymaga zatem użycia większej (min. 2) liczby instrukcji i zajmuje odpowiednio więcej miejsca w pamięci oraz czasu (cykli zegarowych).

Działania na liczbach bez znaku

- **poprawność obliczeń**
- **przekroczenie zakresu**

przykład: **odejmowanie (dodawanie uzupełnienia) półbajtów** (zakres 0 -15):

$C_{MSB}=1$

```
  0101 (5)
  1111 (-1)
+ -------
  0100 (4)
```

$C_{MSB}=0$

```
  0101 (5)
  1001 (-7)
+ -------
  1110 (?? 14)
```

„Surowe” (wychodzące bezpośrednio z ALU) przeniesienie z najstarszej pozycji (MSB) podczas odejmowania zachowuje się **odwrotnie** niż podczas dodawania:

- wystąpienie tego przeniesienia oznacza prawidłowy wynik,
- jego brak – przekroczenie zakresu.

- W powyższy sposób ustawiana jest flaga CARRY w np. procesorach ARM,

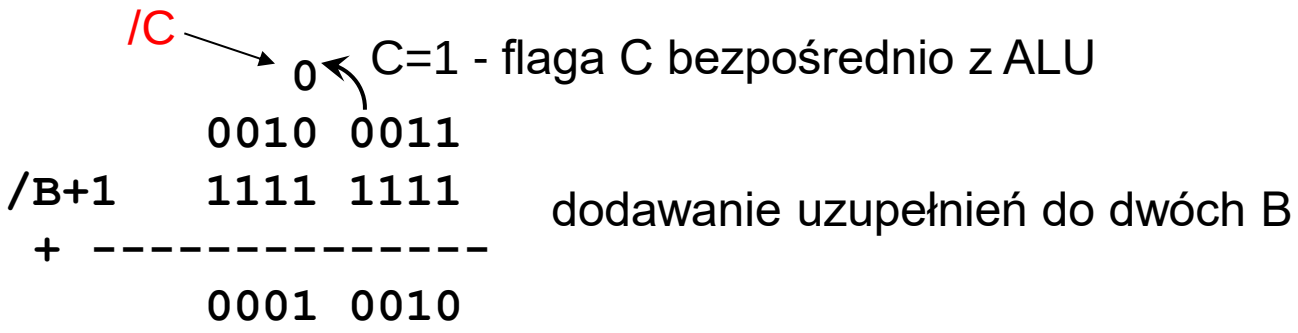
- w innych rozwiązaniach (**x86**, AVR) **przeniesienie po odejmowaniu jest negowane**, (flaga CARRY zachowuje się tak, jak w przypadku dodawania).

SBC - Subtract with CARRY (ARM)

Przykład (4-bitowy):

35 – 17

```
          H       L
A       0010    0011    (35)
B       0001    0001    (17)
    - --------------
        ????    ????
```



pierwsza część (najmłodsza):

$A_L - B_L = A_L + /B_L + 1$

druga (i kolejne części):

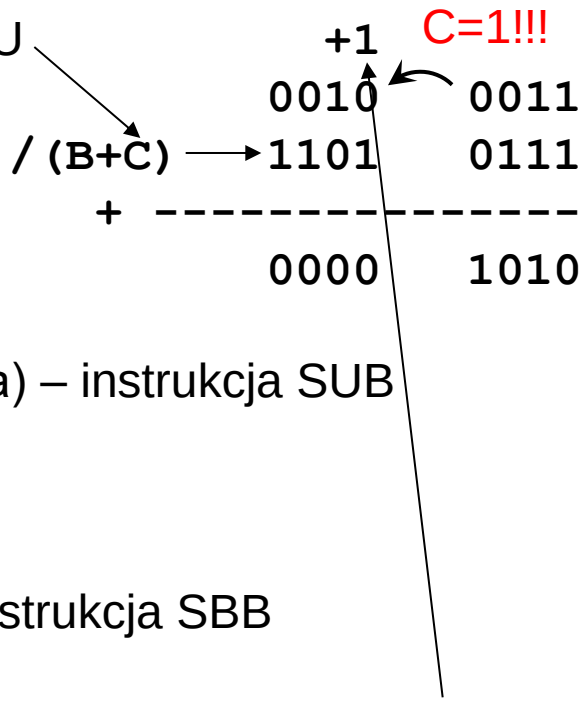
$A_H - B_H + /C = A_H + /B_H + 1 + /C$

Subtract with BORROW (x86, AVR)

przykład:
35 - 25

```
          H       L
A       0010    0011    (35)
B       0001    1001    (25)
    - ------------
        ????    ????
```

C = 1 - zanegowane przeniesienie z ALU



pierwsza część (najmłodsza) – instrukcja SUB

$A_L - B_L = A_L + /B_L + 1$

druga (i kolejne części): - instrukcja SBB

$A_H - B_H - C = A_H - (B_H + C) = A_H + /(B_H + C) + 1$

Działania na liczbach ze znakiem (U2)

- **poprawność obliczeń**
- **przekroczenie zakresu**

Przykłady obliczeń na **półbajtach** (zakres: -8 …+7):

$C_{MSB-1}=0$, $C_{MSB}=0$

```
  0 0 1 0  (2)
  1 1 0 1 (-3)
+ ----------
  1 1 1 1 (-1) OK
```

$C_{MSB-1}=1$, $C_{MSB}=1$

```
  1 1 1 0 (-2)
  1 1 0 1 (-3)
+ ----------
  1 0 1 1 (-5) OK
```

$C_{MSB-1}=1$, $C_{MSB}=0$

```
  0 0 1 1 (3)
  0 1 1 0 (6)
+ ----------
  1 0 0 1 (-7) ?? 9 ?? - nadmiar!
```

$C_{MSB-1}=0$, $C_{MSB}=1$

```
  1 1 0 1 (-3)
  1 0 1 0 (-6)
+ ----------
  0 1 1 1 (7) ?? - nadmiar!
```

W przypadku dodawania (odejmowania) liczb ze znakiem (U2) **przekroczenie zakresu sygnalizowane jest ustawieniem flagi OVERFLOW** obliczanej jako:
$OVERFLOW=C_{MSB}$ XOR C_{MSB-1}

Flagi (arytmetyczne) procesora

• bity w dedykowanym rejestrze procesora (rejestrze flag, statusu, stanu itp.).

Generalnie* ustawiane na podstawie wyniku każdej operacji arytm./ logicznej,

• w praktyce: – sprawdzić w dokumentacji procesora!

np. **instrukcje INC / DEC nie ustawiają flagi CARRY** (x86, AVR).

Instrukcje porównująco-testujące (np. CMP i TEST w x86) ustawiają tylko flagi, nie zapisują nigdzie wyniku!

Na podstawie stanu flag są wykonywane (ignorowane) skoki warunkowe.

* w niektórych architekturach (np. ARM-Cortex) instrukcje arytmetyczne domyślnie nie ustawiają flag. Ustawieniem steruje pole bitowe w kodzie rozkazu.

Flagi (arytmetyczne) procesora

- procesor nie posiada dedykowanych instrukcji do dodawania/odejmowania liczb ze znakiem i bez znaku (w przeciwieństwie np. do mnożenia/dzielenia),

- pobiera jedynie „ciągi zer i jedynek” (o danej długości) i przetwarza je w ściśle określony sposób,

- oraz **ustawia wszystkie* flagi arytmetyczne**:

CARRY – przeniesienie (liczby bez znaku)
OVERFLOW – nadmiar (ze znakiem)
SIGN = bit znaku = najstarszy bit (MSB)
ZERO (stan wysoki gdy wynik operacji wynosi zero)
AUX CARRY – przeniesienie pomocnicze (między młodszym a starszym półbajtem)
PARITY – wskazuje, czy liczba jedynek w najmłodszym bajcie jest parzysta

- **od programisty zależy interpretacja wyników i sens używania poszczególnych flag** (np. przy skokach warunkowych).

* w przypadku instrukcji typu ADD / ADC / SUB / SBB.
Natomiast instrukcje INC/DEC nie modyfikują CARRY!

Przesunięcia bitowe

logiczne - bitowe / liczby bez znaku

np. zmienna 8-bitowa: 1001 1110
w lewo o 1 bit Carry < **1** < 0011 110**0**
w prawo o 1 bit **0**100 1111 > **0** > Carry

• **na „zwolnione" miejsce wpisywane jest zawsze zero!**

W x86 instrukcje: SHL/SHR $liczba_bitów , rejestr/pamięć

arytmetyczne / liczby ze znakiem

np. zmienna 8-bitowa: 1001 1110
w lewo o 1 bit Carry < **1** < 0011 110**0** (zmiana znaku = nadmiar, jak „+" U2)
w prawo o 1 bit **1**100 1111 > **0** > Carry

• w lewo – działa tak, jak przesunięcie logiczne,
• **w prawo: powielona poprzednia wartość najstarszego bitu (zachowuje znak).**

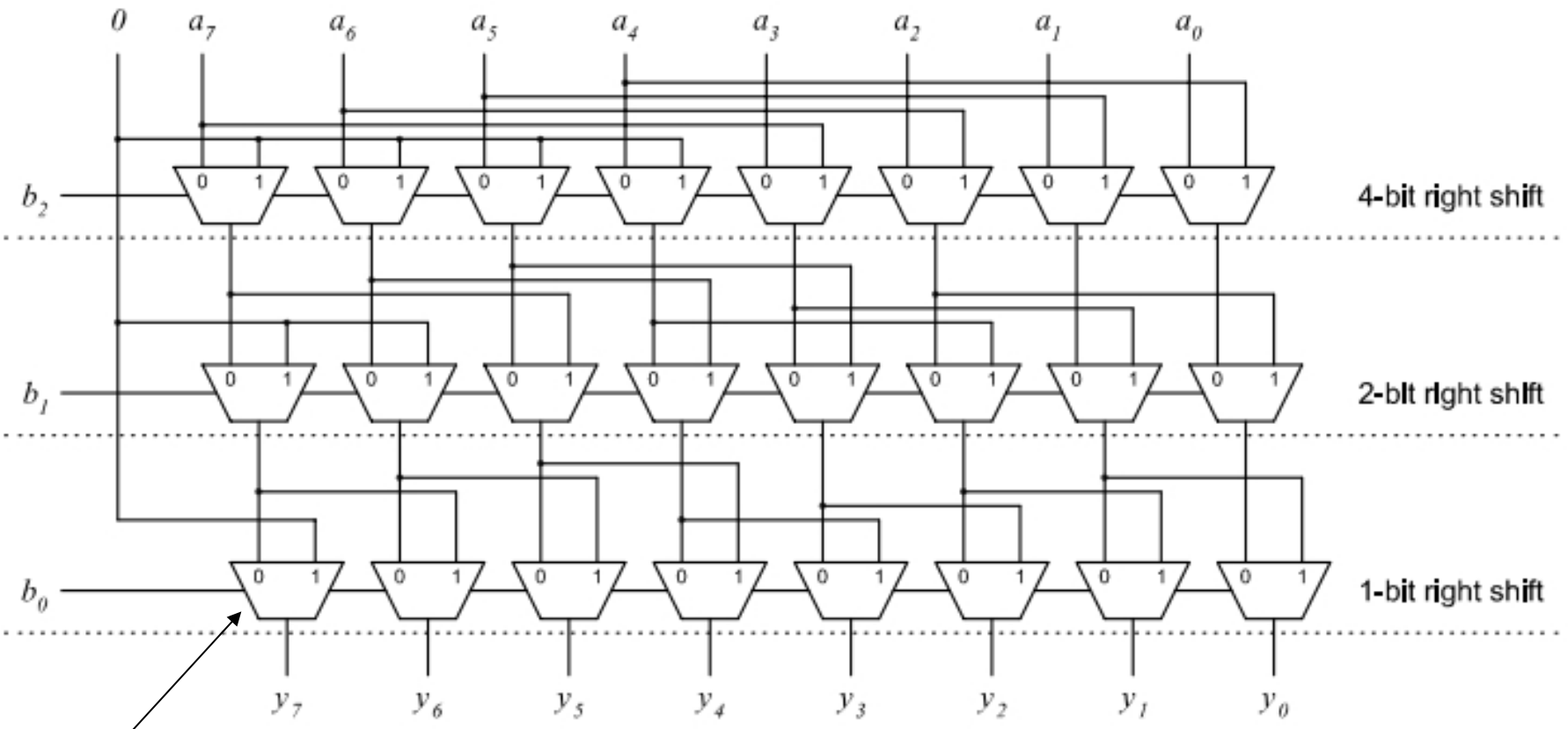
W x86 instrukcje: SAL/SAR $liczba_bitów , rejestr/pamięć.

Przesunięcia bitowe to szybkie mnożenie/dzielenie przez stałą - potęgę 2.

Przesunięcia bitowe

Barell shifter – szybki przesuwnik bitowy:

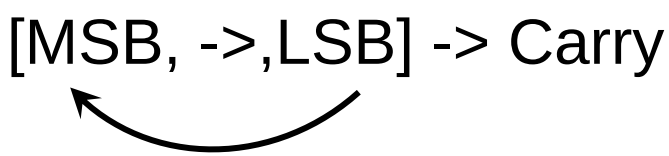
czas wykonania operacji nie zależy od liczby przesuwanych miejsc.



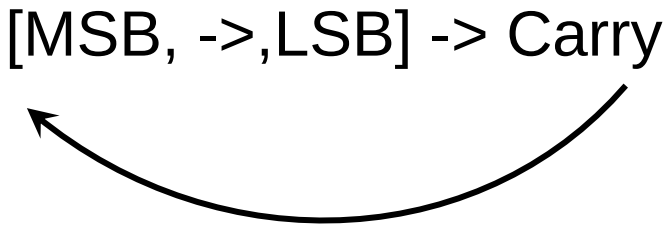
MUX – multiplekser – selektor wejścia: if b=0 then out = in0 else out = in1

Obroty bitowe

„zwykłe” w lewo i w prawo (ROtate Left/Right)
ROR/ROL $liczba_bitów , rejestr/pamięć



poprzez flagę przeniesienia – Rotate through Carry Left/Right
RCR/RCL $liczba_bitów , rejestr/pamięć



Rozszerzenie długości słowa 1/4

np. z 4 na 8 bitów:

bez znaku - ZERO EXTEND

0101 -> 0000 0101 – uzupełnienie bardziej znaczącej części zerami

ze znakiem – SIGN EXTENT

0101 -> **0000** 0101 (5)
1101 -> **1111** 1101 (-3)

uzupełnienie bardziej znaczącej części zgodnie ze znakiem liczby!

w x86 instrukcje typu MOVZX, MOVSX…

Rozszerzenie długości słowa 2/4

Architektura x86–64, rejestry 64-bitowe

• operacje 8- i 16-bitowe np. mov nie modyfikują starszych, niewykorzystanych części rejestrów:

```
.data
int64: .quad 0

.text
MOV int64 , %rax      %rax = 0x0000000000000000 (64 zera)
NOT %rax              %rax = 0xFFFFFFFFFFFFFFFF (64 jedynki)
MOV $0,%ah            %rax = 0xFFFFFFFFFFFF00FF
```

• operacje 32-bitowe zerują starszą część rejestru.

```
XOR $eax,%eax         %rax =  0x0000000000000000 (64 zera)
NOT %rax              %rax =  0xFFFFFFFFFFFFFFFF (64 jedynki)
MOV $0,%eax           %rax =  0x0000000000000000
```

Rozszerzenie długości słowa 3/4 - ładowanie stałych do rejestru 64-bitowego, przekazywanych bezpośrednio w kodzie rozkazu (immediate).

Instrukcja zapisana w pliku *.s -> tłumaczenie podczas kompilacji (*as*), plik *.o

```
MOV $0x00 , %rax                    MOV $0x00000000, %rax
                                    # rax = 0x0000000000000000

MOV $0x7FFFFFFF , %rax              MOV $0x7FFFFFFF , %rax
                                    # rax = 0x000000007FFFFFFF
```

stałe zostały zakodowane jako **32-bitowe rozszerzane do 64 bitów z zachowaniem znaku** (tutaj MSB=0). Jednak gdy spróbujemy załadować 32-bitową z MSB=1 (np. ujemną), co powinno skutkować:

```
MOV $0x8FFFFFFF , %rax              MOV $0x8FFFFFFF , %rax
                                    # rax = 0xFFFFFFFF8FFFFFFF
```

kompilator *GAS* użyje rozkazu MOVABS, przekazując pełną, 64-bitową stałą:

```
MOV $0x8FFFFFFF , %rax      ->      MOVABS $0x000000008FFFFFFF,%rax
```

Przekazanie (w pliku źródłowym .s) dowolnej 64-bitowej stałej odbywa się w sposób:

```
MOV $0xFFEEDDCC99887766,%rax ->   MOVABS $0xFFEEDDCC99887766,%rax
```

Rozszerzenie długości słowa 4/4

Inne operacje, np. arytmetyczno-logiczne z rejestrem 64-bitowym umożliwiają użycie tylko **stałej 32-bitowej, rozszerzanej do 64 bitów z uwzględnieniem jej znaku**.

Należy zwrócić uwagę na składnię w kompilatorze *GAS*:

```
XOR   %eax,%eax                    %rax = 0x0000000000000000
OR    $0x80000000,%rax
```

Stała 32-bitowa w rozkazie OR ma MSB=1, co zgodnie z rozszerzeniem bitu znaku sugerowałoby następującą operację na pełnym rejestrze 64-bitowym:

```
OR    $0xFFFFFFFF80000000,%rax
```

w efekcie której otrzymalibyśmy: `%rax = 0xFFFFFFFF80000000`

Taki zapis rozkazu (32-bitowa stała z MSB=1 i 64-bitowy rejestr) jest niedopuszczalny w *GAS*. Poprawny zapis ww. operacji w „*as*” to: `OR $0xFFFFFFFF80000000,%rax`

Poniższe zapisy sum logicznych są również poprawne (zastanowić się – czemu?):

```
                                   wynik operacji:
XOR   %eax,%eax                    %rax = 0x0000000000000000
OR    $0x70000000,%rax             %rax = 0x0000000070000000
OR    $0x80000000,%eax             %rax = 0x00000000F0000000
```

Operacje na bitach

testowanie (sprawdzanie) bitu znajdującego się na danej pozycji

```
      0011 0110
      0001 0000 – maska zer i jedynki na testowanej pozycji
AND ----------
      0001 0000 - wynik=waga testowanej pozycji gdy dany bit=1
                  wynik=0 gdy bit=0
```

W x86 oprócz instrukcji **AND** istnieje również rozkaz:

TEST argument1 , argument2

Jest to iloczyn logiczny bez zapisywania wyniku, ustawiane są tylko flagi procesora (nie nadpisuje drugiego argumentu – podobnie jak CMP).

Operacje na bitach

zerowanie bitu na danej pozycji

```
      0011 0110
      1110 1111 – maska jedynek i zera na pozycji kasowanej
AND ----------
      0010 0110
```

ustawianie bitu na danej pozycji

```
      0011 0110
      0000 1000 – maska z zer i jedynki na ustawianej pozycji
 OR ----------
      0011 1110
```

można stosować dla grup kilku bitów…

Operacje na bitach

zmiana wartości bitu na danej pozycji **na wartość przeciwną**

0011 0110
1111 0000 – jedynki w masce – zmiana wartości na przeciwną
XOR ---------- zera – pozostawienie bez zmian
1100 0110

W x86 istnieją również rozkazy typu BitTest:

BT, BTC, BTS, BTR …

umożliwiające testowanie i zmianę wskazanego bitu.

W wielu mikrokontrolerach istnieją dedykowane rejestry (bądź wręcz adresowalne pojedyncze bity), pozwalające wykonywać ww. operacje w sposób „atomowy”.

Mnożenie liczb całkowitych (bez znaku)

```
                1011 (11)
                0101 (5)
              x -------
                1011
               0000
              1011
             0000
           +-----------
MSB=Carry=0  00110111 (55)
```

dane: *n*-bitowe argumenty *a* - mnożna, *b* - mnożnik, **iloczyn 2*n* bitów:** *p*=0

for (*i*=0; *i*<n; *i*++)

{

testuj *i*-ty bit w *b*: jeśli bit=1 to *p:=p*+*a*

przesuń *a* o 1 bit w lewo

}

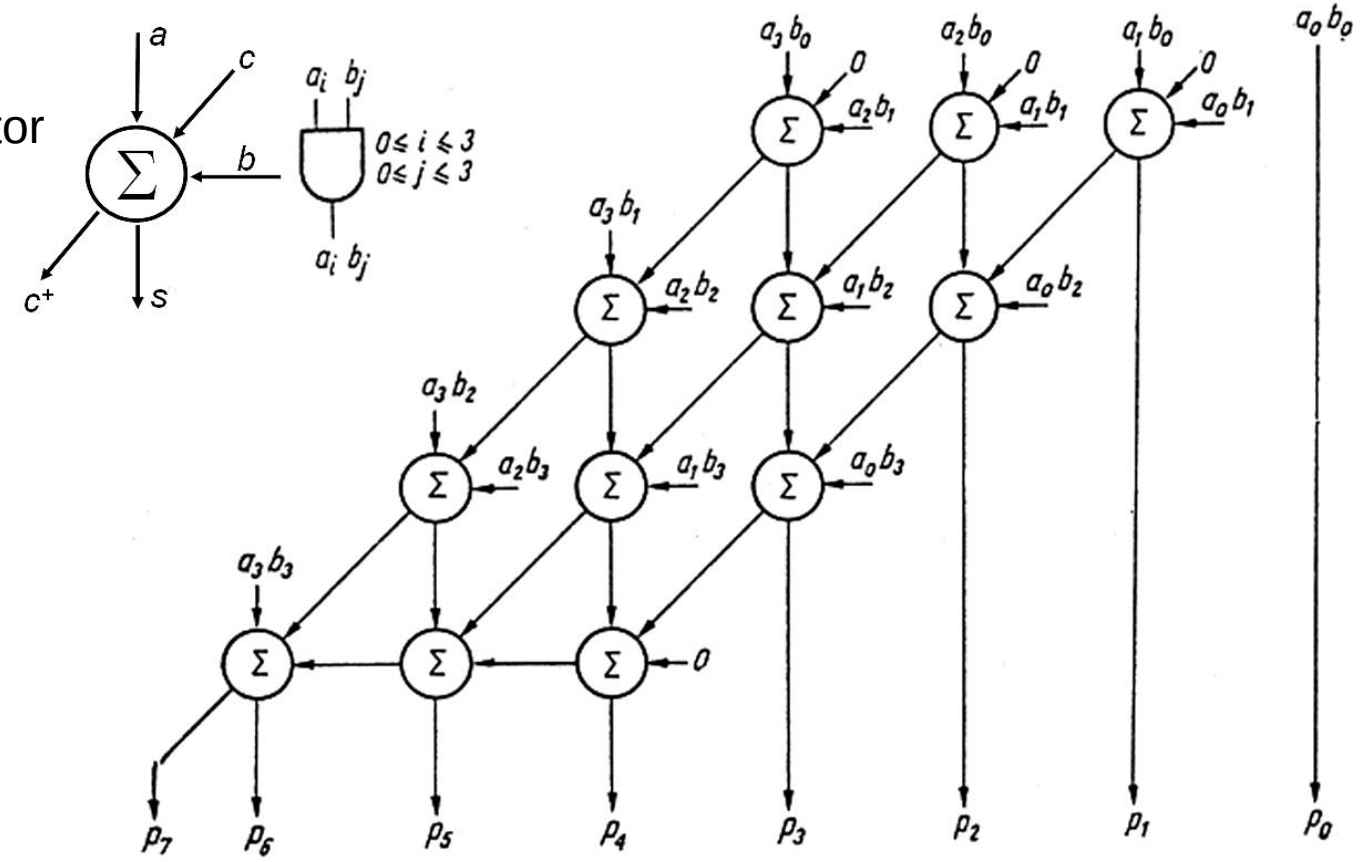
- proste, ale liczba iteracji (cykli zegarowych - tym samym czas całej operacji) zależy od długości mnożonych słów…

Mnożenie liczb całkowitych (bez znaku)

mnożący układ macierzowy

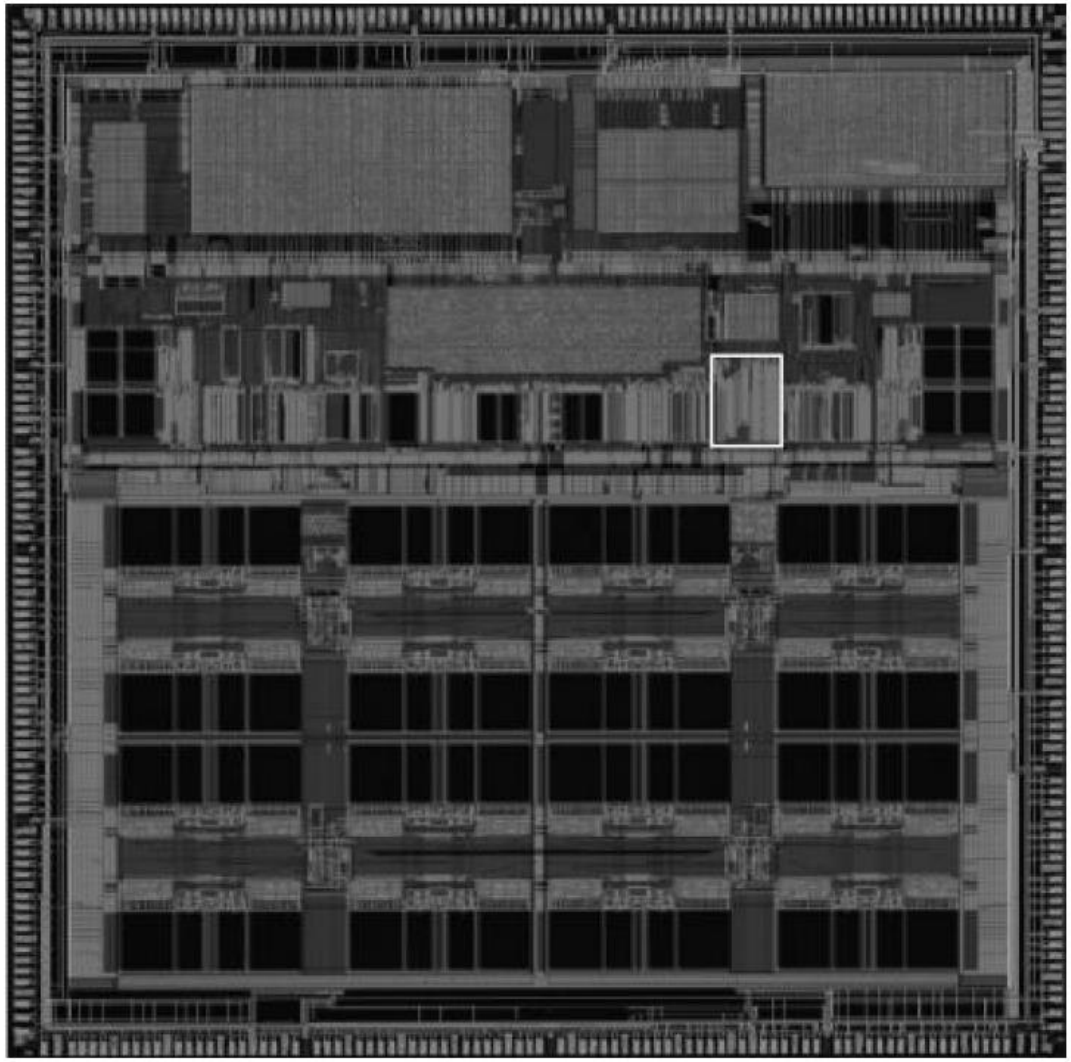
- zasada działania – mnożenie „w słupku”…,
- zaleta: duża szybkość, (ograniczona czasem propagacji przez bramki logiczne w najdłuższej ścieżce sygnałowej).

sumator pełny:



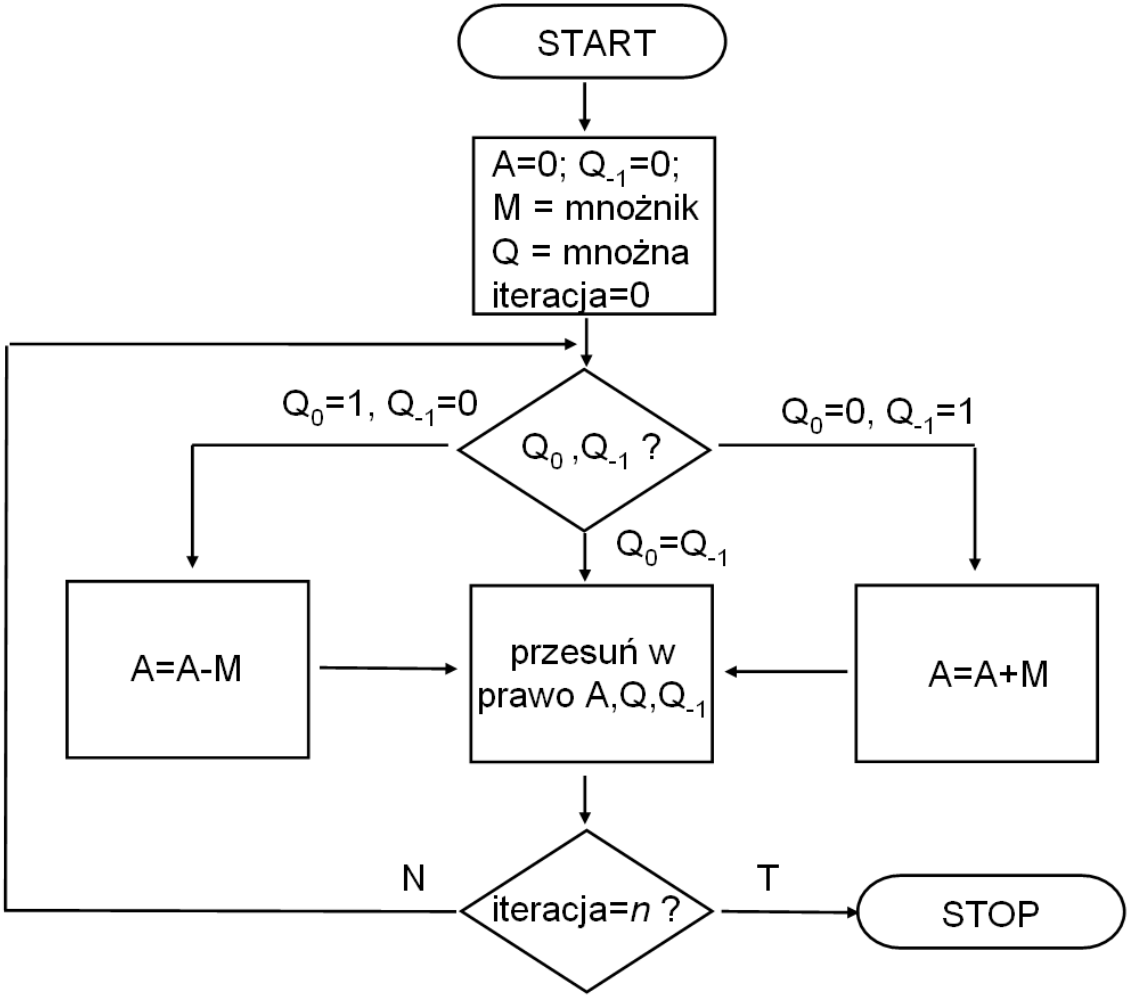
Struktura procesora ARM10200

szerokie (np. 32-bitowe) macierzowe układy mnożące „zajmują” stosunkowo dużą powierzchnię w strukturze całego układu…



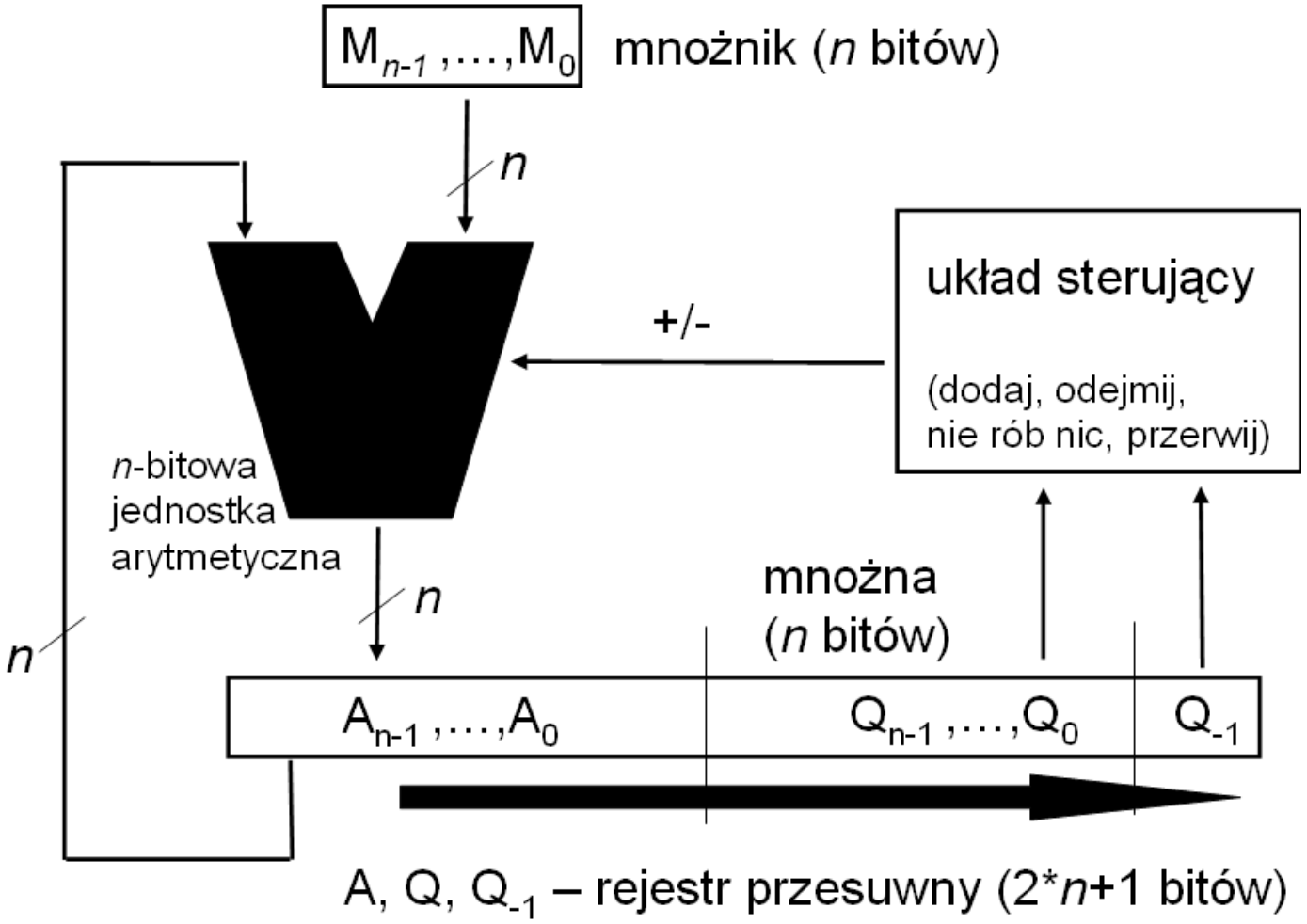
algorytm Bootha - mnożenie liczb całkowitych ze znakiem

dane: mnożna *Q* i mnożnik *M*, każdy argument ma *n* bitów,
rejestr przesuwny - trzy części: *A* i *Q* po *n* bitów, Q_{n-1} – 1 bit (razem 2*n*+1 bitów),
wynik po *n* iteracjach w części *A* i *Q* (2*n* bitów).



algorytm Bootha

Mając w procesorze ALU z możliwością + i – , można stosunkowo prosto zbudować układ mnożący w/w algorytmem…



Przykład -3 x 7 (argumenty 4-bitowe)

```
     A    Q Q-1
  0000 1101 0       1 i 0  - odejmij mnożnik
  1001              -7
+-------
  1001 1101 0
-> SAR              przesunięcie arytmetyczne w prawo!
  1100 1110 1       0 i 1 - dodaj mnożnik
  0111              +7
+-------
  0011 1110 1
-> SAR
  0001 1111 0
  1001              -7
+-------
  1010 1111 0
-> SAR
  1101 0111 1       ostatnie bity jednakowe
-> SAR
  1110 1011  (-21)
```

Restoring division - dzielenie liczb całkowitych bez znaku

dane: *a* - dzielna, *b* – dzielnik,
rejestr przesuwny: dwie części *A* i *Q* po *n* bitów,
wynik: reszta *r* z dzielenia (modulo) w *A*, iloraz *q* w *Q.*

wykonywane działanie:

$$\frac{a}{b} = q + \frac{r}{b}$$

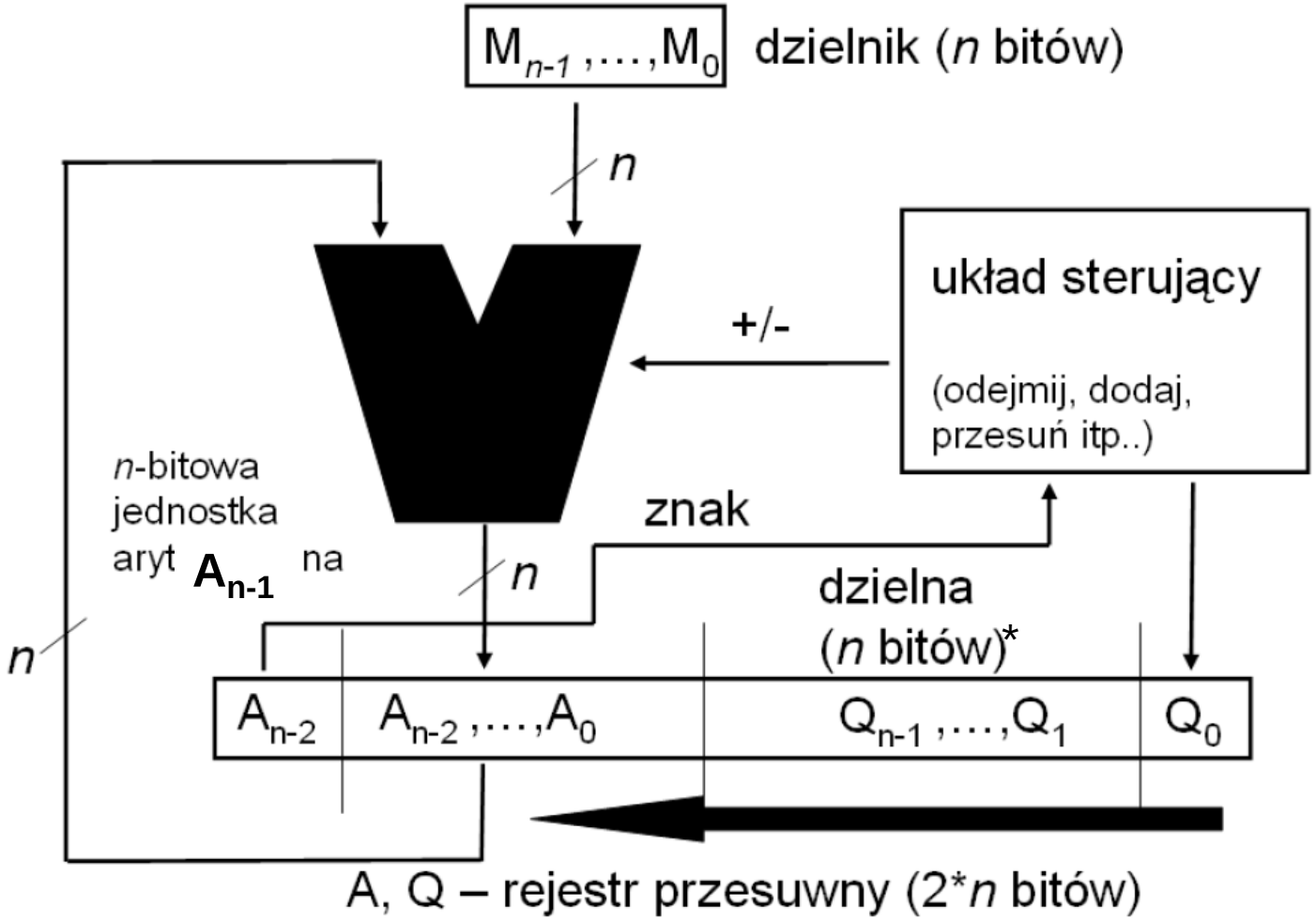
inicjowanie zmiennych:
A = 0 (akumulator, *n* bitów)*
M = *b* (dzielnik, *n* bitów)
Q = *a* (dzielna, *n* bitów)*

for (*i*=0; *i*<n; *i*++)
{
    przesuń *A* i *Q* o 1 bit w lewo
    oblicz *A*:=*A*-*M*
    jeśli *A*<0 to ustaw Q_0=0 i przywróć (restore) poprzednią wartość *A* (*A*:=*A*+*M*),
            w przeciwnym razie ustaw Q_0=1.
}

*dzielna może mieć 2*n* bitów (w A i Q), pod warunkiem, że iloraz da się zapisać w *n* bitowym Q

Restoring division - dzielenie liczb całkowitych bez znaku

Podobnie jak w przypadku alg. Bootha, mając w procesorze ALU stosunkowo niedużym kosztem można dobudować układ dzielący.



Przykład 11 / 3 (argumenty 4-bitowe)

```
     A    Q
  0000 1011 A=0 Q=11
  <- shl - 1. iteracja
  0001 0110
  1101 (-3)
+-------
  1110 (<0, przywróć poprzednie A, Q0=0)

  0001 0110
  <- shl -  2. iteracja
  0010 1100
  1101 (-3)
+-------
  1111 (<0, przywróć poprzednie A, Q0=0)

  0010 1100
  <- shl - 3. iteracja
  0101 1000
  1101 (-3)
+-------
  0010 (>0, pozostaw aktualne A, Q0=1)

  0010 1001
  <- shl - 4. iteracja,
  0101 0010
  1101 (-3)
+-------
  0010 (>0, pozostaw i Q0=1)

  0010 0011
```

Wynik:

A = 2 reszta

Q = 3 iloraz

Mnożenie i dzielenie liczb całkowitych we współczesnych procesorach

- W przeciwieństwie do **+** i **–** procesory posiadają dedykowane instrukcje do ***** i **/** liczb ze znakiem i bez znaku.

- Większość współcześnie projektowanych/produkowanych procesorów (również sygnałowych, graficznych i mikrokontrolerów) pozwala wykonać mnożenie w jednym (góra kilku) cyklu zegarowym, ponieważ iloczyny cząstkowe mogą być wyznaczone jednocześnie.

- Dzielenie jest generalnie operacją złożoną – iteracyjną.
  W przedstawionym (prostym…) algorytmie kolejne etapy (*restore* i wyznaczenie bitów: ilorazu i reszty) zależą od wyniku (znaku) poprzedzającego je odejmowania. Nie można ich zatem (w prosty sposób)* wykonać równolegle (jak w mnożeniu).

- Liczba cykli/mikrooperacji zależy nie tylko od implementowanego algorytmu dzielenia, ale również od rozmiaru danych i konkretnych wartości dzielnej i dzielnika. Dodatkowo, liczba cykli może się znacznie różnić między kolejnymi generacjami tej samej architektury (np. x86-64).

*w praktyce, w mikroprocesory często wykonują dzielenie w oparciu np. o algorytm SRT, możliwe jest wyznaczenie więcej niż jednego bitu w jednej iteracji.

Mnożenie i dzielenie liczb całkowitych we współczesnych procesorach

Liczba cykli zegarowych, potrzebnych do wykonania * i /
w 32-bitowym mikrokontrolerze ARM-Cortex M4:

Multiply	Multiply	MUL Rd, Rn, Rm	1	
	Multiply accumulate	MLA Rd, Rn, Rm	1	
	Multiply subtract	MLS Rd, Rn, Rm	1	
	Long signed	SMULL RdLo, RdHi, Rn, Rm	1	
	Long unsigned	UMULL RdLo, RdHi, Rn, Rm	1	
	Long signed accumulate	SMLAL RdLo, RdHi, Rn, Rm	1	
	Long unsigned accumulate	UMLAL RdLo, RdHi, Rn, Rm	1	
Divide	Signed	SDIV Rd, Rn, Rm	2 to 12	Division operations terminate when the divide calculation completes, with the number of cycles required dependent on the values of the input operands. Division operations are interruptible, meaning that an operation can be abandoned when an interrupt occurs, with worst case latency of one cycle, and restarted when the interrupt completes.
	Unsigned	UDIV Rd, Rn, Rm	2 to 12	

Zestawienia liczby mikrooperacji, niezbędnych do wykonania rozkazów
w architekturach x86 różnych generacji są dostępne na stronie:

https://www.agner.org/optimize/instruction_tables.pdf